RICE UNIVERSITY

Static Cost Estimation for Data Layout Selection on GPUs

by

Yuhan Peng

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

Vivek Sarkar, Chair
Professor of Computer Science
E.D. Butcher Chair in Engineering

Keith D. Cooper
L. John and Ann H. Doerr Professor of Computational Engineering
Professor of Computer Science

John Mellor-Crummey
Professor of Computer Science
Professor of Electrical and Computer Engineering

Houston, Texas

April, 2017
ABSTRACT

Static Cost Estimation for
Data Layout Selection on GPUs

by

Yuhan Peng

Performance modeling provides mathematical models and quantitative analysis for designing and optimizing computer systems and architectures. For many data-intensive applications, high-latency memory accesses often dominate execution time. Thus, performance modeling for memory accesses on high performance architectures has become an important research topic.

The data layout of an application refers to the way in which data is stored and organized. In high performance computation, the data layout can significantly affect the efficiency of memory access operations. In recent years, the problem of data layout selection has been well studied on various multi-core CPU and some heterogeneous architectures.

GPUs have memory hierarchies different from multi-core CPUs. While data layout selection on GPUs has been studied previously, none of the prior work provides a mathematical cost model for data layout selection on GPUs. This motivates us to investigate static cost analysis methods to guide data layout selection work, and perhaps even the design of new SIMT architectures.

This thesis presents a comprehensive cost analysis for data layout selection on GPUs. We build our cost function based on knowledge of the GPU memory hierarchy,
and develop an algorithm which enables researchers to perform compile time cost estimation for a given data layout. Furthermore, we introduce a new vector based cost representation of the estimated cost, which can better estimate the memory access cost of applications with dynamic length loops. We apply our cost analysis to benchmarks considered by prior work on data layout selection, and our experimental results show that our cost analysis can accurately predict the relative costs of different data layouts.
Acknowledgements

I would like to express my great appreciate to my advisor, Prof. Vivek Sarkar, for his great help advising my research and guidance in writing this thesis. I also thank Prof. Keith Cooper and Prof. John Mellor-Crummey for being my thesis committee and providing feedback for my future research.

Moreover, I would like to thank Max Grossmam for his help in developing the cost model and proofreading the thesis. I would like to thank all other members in the Habanero Extreme Scale Software Research Group and the members from the Center for Domain-Specific Computing at UCLA, for their help and reference for my research.

Finally, I would like to thank my family members and friends for their love and support, especially when I have difficulties with my research and life.

The work in this thesis is partially supported by the Intel Corporation with the matching fund from the NSF under the Innovation Transition (InTrans) Program (CCF-1436827).
## Contents

Abstract ii  
List of Illustrations viii  
List of Tables ix

1 Introduction 1  
1.1 Motivation .................................................. 1  
1.2 Thesis Statement ............................................ 2  
1.3 Contributions ............................................... 2  
1.4 Thesis Roadmap ............................................. 3

2 Background 4  
2.1 GPU Memory Hierarchy ..................................... 4  
2.2 GPU Memory Access ........................................ 6  
2.3 GPU Memory Access Profiling Metrics .................... 7  
2.4 The Impact of Different Data Layouts .................... 9  
2.5 Automatic Data Layout Selection ......................... 13

3 Dynamic Cost Functions 15  
3.1 Assumptions .................................................. 15  
3.2 Design of the Cost Functions ............................... 16  
3.3 Deriving the Dynamic Cost from Profiling Metrics ........ 17  
3.4 Illustration ................................................. 18

4 Static Cost Estimation 21
4.1 Assumptions ................................................................. 21
4.2 Basic Terminology for Accessing Global Memory ................. 22
  4.2.1 General Structure of SoAoS ....................................... 22
  4.2.2 Block Size, Grid Size, and Thread ID .......................... 23
  4.2.3 Structure Size ..................................................... 23
  4.2.4 Array Index ....................................................... 24
  4.2.5 Stride ............................................................ 24
  4.2.6 Instruction Distance ............................................. 26
  4.2.7 Index Accordance ................................................. 27
4.3 Estimating the Cost Coefficient ........................................ 27
4.4 Estimating the Number of Transaction Requests ..................... 28
4.5 Estimating the Cache Hit Ratio ....................................... 29
4.6 Estimating the Number of Blocks per SM ............................ 31
4.7 Estimating the Total Cost ............................................. 32
4.8 Handling Dynamic Length Loops ..................................... 35

5 Performance Evaluation ................................................. 39
  5.1 Performance Evaluation Methodology ............................... 39
  5.2 Machine Configuration .............................................. 39
  5.3 Benchmarks .......................................................... 40
    5.3.1 K-means ........................................................ 40
    5.3.2 N-body ......................................................... 40
    5.3.3 LavaMD ......................................................... 41
    5.3.4 Summary ....................................................... 41
  5.4 Performance Results ................................................ 41
  5.5 Discussion .......................................................... 44

6 Related Work ............................................................ 46
  6.1 Cost and Data Layout Analysis for CPUs ........................... 46
6.2 Cost Analysis for GPUs .................................................. 46
6.3 Data Layout Analysis for GPUs ........................................... 47
6.4 Automatic Data Layout Selection on GPUs ......................... 48
  6.4.1 Distance Based Approaches ....................................... 48
  6.4.2 Other Approaches .................................................. 50

7 Conclusions and Future Work ........................................... 51

Bibliography ................................................................. 52
Illustrations

2.1 The GPU memory hierarchy. Figure source: NVIDIA CUDA 0.8 SDK. [1] .................................................. 5

2.2 The hierarchy of GPU memory access profiling metrics, modified from [2]. ........................................... 10

2.3 The memory organization of our sample AoS, SoA, and SoAoS with N = 100. ........................................... 12

4.1 One example of different strides. ................................................................. 25

4.2 Dependency diagram of the attributes for cost estimation. ........... 33

5.1 Performance result on an Nvidia Fermi GPU, normalized to the AoS layout. .................................................. 43

5.2 Performance result on an Nvidia Kepler GPU, normalize to the AoS layout. .................................................. 43

6.1 The cost function proposed by Kofler et al. Figure source: [3]. ....... 48

6.2 The cost function proposed by H2C. Figure source: [4]. ................. 49
Tables

3.1 Deriving the value of the terms in the cost function from the profiling metrics. .......................................................... 18

3.2 The value of the profiling metrics, the dynamic cost, as well as the execution time of the two-field vector addition example on an Nvidia Fermi GPU. ........................................................... 19

4.1 Latencies for the memory hierarchy in an Nvidia Fermi GPU. .... 28

5.1 Attributes of the GPUs used for evaluation. .......................... 40

5.2 Benchmarks and data layouts for our performance evaluation .... 42

5.3 Performance results on selected benchmarks and data layouts on Nvidia Fermi and Kepler GPUs, relative to the AoS layout. .... 44
Chapter 1

Introduction

1.1 Motivation

Currently, GPUs play an important role in the world of parallel computing. Compared to multi-core CPUs, GPUs have a larger number of computational cores and are able to better handle heavy computational tasks. However, the latency of memory access operations is significantly higher than arithmetic operations on GPUs, which means a memory access pattern with poor locality can lead to greater performance loss on GPUs than on CPUs. Thus, improving the efficiency of memory access operations is critical on GPUs.

The data layout of an application refers to the manner in which data is stored and organized, and the choice of data layout can significantly affect the efficiency of memory accesses on the GPU. Moreover, as a result of the unique architectural features of the GPU, for a given application the best data layout on the GPU is usually different from the best data layout on the CPU.

Part of the reason why determining the best data layout for a GPU kernel is difficult is because the number of possible data layouts can be numerous, and finding the best data layout has been proven to be NP-hard [5, 6]. In many cases, the best data layout is not intuitive. For that reason, it would be ideal to select the best data layout automatically as part of compile-time optimization. In recent years, the automatic data layout selection problem has been studied on GPUs. Unfortunately,
none of the existing approaches proposes a model to estimate the cost for a given data layout. Without a cost model, the performance of a data layout selection algorithm can only be evaluated from the performance results of selected benchmarks, which might be misleading. This motivates us to build a unified cost model that can be applied to all data layout selection algorithms.

In this thesis, we introduce a comprehensive mathematical cost model based on the GPU memory hierarchy, as well as an algorithm to estimate the cost of a given data layout at compile time. We also propose a novel cost vector representation to better handle cost estimation in the presence of dynamic length loops. Our cost model is not limited to data layout selection among different combinations of structures and arrays, and can be applied to any data layout selection problem on GPUs. The goal is to build an accurate cost model for data layout selection on GPUs, where data layouts with lower costs will have better execution efficiency.

1.2 Thesis Statement

It has been established in past work that data layouts can have a significant impact on GPU performance. Our thesis is that it is possible to use static cost estimation to effectively determine when one data layout will perform better than another, thereby making it possible to statically choose the best or near-best layout from a set of different data layouts for a given GPU kernel.

1.3 Contributions

This thesis makes the following contributions:

- We introduce a cost function for accurately modeling the cost of executing GPU
kernels with different data layouts.

- We design a cost estimation algorithm which can estimate the cost of a given data layout for a GPU kernel at compile time, i.e., without executing the kernel.
- We propose a vector based representation for better estimating the static cost of kernels with dynamic length loops.

1.4 Thesis Roadmap

Chapter 2 introduces on the basic terminology and concepts of the GPU computation model and data layout to aid in understanding our approach for cost analysis. Chapter 3 introduces the design of our dynamic cost function. Chapter 4 describes our static cost estimation algorithm based on the GPU kernel and a given data layout. Chapter 5 describes the application of our cost estimation algorithm to selected benchmarks and compares the estimated cost with actual performance. Chapter 6 discusses related work on GPU performance evaluation and automatic data layout selection. Chapter 7 summarizes our conclusions and the future work of this project.
Chapter 2

Background

2.1 GPU Memory Hierarchy

GPUs have a unique memory hierarchy which includes both on-chip memory and off-chip memory. In general, on-chip memories are smaller but faster than off-chip memories. Figure 2.1 shows the GPU memory hierarchy before the introduction of GPU caches. As shown in the figure, the on-chip memory consists of registers and shared memory, while the off-chip memory consists of global memory, constant memory and texture memory.

In order to improve global memory transaction efficiency, modern GPU models have added both L1 and L2 caches. For example, the Fermi GPU has 64KB configurable shared memory and L1 cache, as well as 768KB unified L2 cache [7]. Later models like Kepler and Maxwell further enlarge the size of the L1 and L2 cache [8, 9]. The on-chip L1 cache has a relatively smaller size, but also a lower latency and higher bandwidth. The L1 cache is shared by all threads inside the same CUDA thread block, where all threads in a block will execute on the same Streaming Multiprocessor (SM). The L1 cache shares the same on-chip memory with CUDA shared memory, and the user can partition the memory space between the L1 cache and shared memory. In contrast, the L2 cache has a larger size, but a much higher latency. The L2 cache is shared by all threads inside the GPU.
Figure 2.1: The GPU memory hierarchy. Figure source: NVIDIA CUDA 0.8 SDK. [1]
2.2 GPU Memory Access

GPUs schedule threads at the granularity of warps, where every warp consists of a set of adjacent threads according to the thread ID. On current Nvidia GPUs, every 32 consecutive threads form a warp. Threads inside a warp share the same program counter, but have their own registers and local memory space [10]. At each cycle, all threads inside the warp may execute one instruction in a Single Instruction, Multiple Thread (SIMT) manner. If there are branches, all threads execute all possible branch paths. For each branch path, threads not going through this path are disabled. After all paths are taken, all threads resume for further execution. The execution of different warps is independent of each other.

Global memory accesses are done in 32, 64 or 128-byte transactions aligned at 32, 64, or 128-byte addresses. When accessing global memory, the warp determines the transaction length according to the type of the access (read or write) and the cache hierarchy of the chip. On some GPU models such as Nvidia Kepler, a 128-byte transaction will be reduced to a 64-byte transaction if only half of the 128 bytes is actually needed, and a 64-byte transaction can be further reduced to a 32-byte transaction [11]. The number of actual transaction requests will be equal to the number of unique memory segments, which means if the threads inside a warp are accessing nearby memory locations, then the memory requests from multiple threads may be satisfiable by a single memory transaction.

Nvidia GPUs combine memory accesses in adjacent threads into fewer memory transactions, and this technique is called memory coalescing [12]. For example, if all threads in the warp are accessing the same memory location, then only one global memory transaction will be needed. If all threads inside a 32-thread warp are accessing adjacent 4-byte memory locations, and the memory location accessed by the
first thread is a multiple of 128 bytes, then only a single contiguous 128-byte memory segment will be accessed, and this is an example of a fully coalesced memory access.

Global memory transactions will go through the L1 and L2 cache. If the L1 cache is present and enabled, then global loads will first look for data in the L1 cache. If a load transaction misses in the L1 cache, it will then look for data in the L2 cache. For current GPUs, global stores are not cached in the L1 cache, as they directly go to the L2 cache. Similarly, the L2 cache is accessed if the memory segment is found in the L2 cache, and GPU Dynamic Random-Access Memory (DRAM) is accessed if the transaction misses in the L2 cache. Memory segments will be cached in both L1 and L2 cache when being loaded from the DRAM.

For this project, we only consider the benefit of the L1 and L2 cache, although GPUs have many special-purpose memories such as shared memory and texture memory. We leave the study of the modeling of these special-purpose memories to future work.

2.3 GPU Memory Access Profiling Metrics

Nvidia GPUs with compute capability 2.0 or above provide hardware performance counters that can be used to measure the performance of GPU kernels with metrics, such as the execution time, the number of memory transactions, the cache hit ratio, etc [13]. Such metrics can be retrieved during execution of the kernel, and they can provide a direct measurement of the efficiency for a given data layout. In this project, we can calculate the dynamic cost of a data layout using the profiling metrics fetched from executing the kernel. The rest of this section discusses GPU memory access profiling metrics relevant to the data layout selection problem.

During the execution of a GPU kernel on an Nvidia GPU with compute capability
greater than xxx, the number of global memory access requests can be retrieved using the profiling metrics \texttt{gld\_transactions} and \texttt{gst\_transactions}. These two metrics show the amount of load and store requests made by all warps, and they only depend on how the accesses are aligned and coalesced, and are independent from the cache hierarchy.

If the L1 cache is present and enabled, then global loads will first look for data in the L1 cache. The number of global loads hits in the L1 cache cannot be retrieved directly, but the profiling metric \texttt{l1\_cache\_global\_hit\_rate} gives the ratio of global loads that hit in the L1 cache. For current GPUs, global stores are not cached in the L1 cache, as they directly go to the L2 cache.

The number of transactions that reach the L2 cache, including both hits and misses, can be retrieved by the metrics \texttt{l2\_l1\_read\_transactions} and \texttt{l2\_l1\_write\_transactions}. Moreover, \texttt{l2\_l1\_read\_hit\_rate} gives the ratio of the global reads hit in the L2 cache, while the ratio of global stores hit in the L2 cache cannot be retrieved directly.

For example, for current GPUs, the L2 cache line size is usually 32 bytes. Then consider one fully coalesced access to a 4 byte data type, leading to a total of 128 adjacent bytes being loaded. Suppose all the transactions were hit in the L2 cache, then for that access, we have \texttt{l2\_l1\_read\_transactions} = 4 * \texttt{gld\_transaction} and \texttt{l2\_l1\_write\_transactions} = 4 * \texttt{gst\_transaction}.

Finally, if a load/store request misses in the L2 cache, then the GPU DRAM will be accessed. We can use the number of transactions reaching the L2 cache and the L2 cache hit ratio to derive the number of transactions reaches the DRAM. \texttt{dram\_read\_transactions} and \texttt{dram\_write\_transactions} do not reflect the total number of DRAM transactions for global memory accesses, because spilling from
local memory accesses also contributes to these metrics [2].

Figure 2.2 shows an overview of the hierarchy of all the GPU profiling metrics about global memory access operations. The red lines indicate data paths through the GPU’s memory hierarchy for servicing global memory transactions, as well as the corresponding profiling metrics for measuring data movement along those paths.

2.4 The Impact of Different Data Layouts

The data layout of an application refers to the way in which data is declared and stored. The choice of data layout can highly impact performance on both CPUs and GPUs, as different data layouts might influence the cache hit ratio, as well as the extent of memory access coalescing. In most cases, the data can be represented as a nested hybrid of arrays (vectors) and structures (objects, classes), and the basic components of a data layout can be either Array-of-Struct (AoS) or Struct-of-Array (SoA).

In AoS, all fields for a logical data point are grouped into the same struct, and one array is declared containing many instances of that struct. The following code shows a sample AoS declaration with data fields \( x \), \( y \) and \( z \).

```c
typedef struct
{
    int x;
    int y;
    int z;
} AoS;
AoS arr[N];
```

In SoA, discrete arrays are declared for different fields. The following code shows a sample SoA declaration, also with data fields \( x \), \( y \) and \( z \).
Figure 2.2: The hierarchy of GPU memory access profiling metrics, modified from [2].
typedef struct
{
    int x[N];
    int y[N];
    int z[N];
} SoA;
SoA arr;

The main difference between AoS and SoA is how the data is organized in memory. In AoS, fields for the same logical data point are in neighboring memory. In SoA, the data for the same field of many logical data points are stored adjacent, but the data for a single data point may be highly strided in memory. Past work has demonstrated that for many applications, the AoS layout can improve CPU cache hierarchy utilization, because different fields for the same logical data point are often accessed together, and in AoS different fields with same array index are declared in the same structure. Accessing one field will also bring the adjacent fields into the cache. On the other hand, SoA may not utilize the cache well, as accessing $x[i]$ will not result in loading $y[i]$ into the cache since $x[i]$ and $y[i]$ are unlikely to be in the same cache line. Therefore, AoS is usually preferred on the CPU.

However, on the GPU, using SoA is more likely to result in coalesced memory accesses, because when all threads in the same warp are accessing the same field for different logical data points simultaneously, using SoA will lead to fewer memory transactions as the same fields are declared in adjacent memory locations. In contrast, AoS might lead to worse coalesced accesses, as declaring different fields in the same struct will waste memory bandwidth when accessing only a single field. Thus, SoA was preferred on the GPU before the GPU cache was introduced.

With the introduction of the GPU’s cache hierarchy, merging closely accessed fields into the same structure may bring back the benefit of cache reuse. As a result,
The best data layout may be some variant of *Struct-of-Array-of-Struct (SoAoS)*. The following example shows a sample SoAoS declaration, designed for the scenario when the data fields \(x\) and \(y\) are accessed with temporal locality, and the data field \(z\) is accessed away from \(x\) and \(y\).

```c
typedef struct {
    int x;
    int y;
} AoS;

typedef struct {
    AoS InnerArr[N];
    int z[N];
} SoAoS;

SoAoS arr;
```

Figure 2.3: The memory organization of our sample AoS, SoA, and SoAoS with \(N = 100\).
Figure 2.3 shows an example of the memory organization of the AoS, SoA and SoAoS declarations above, with \( N = 100 \). It can be seen in AoS, all fields \( x, y \) and \( z \) with the same array index are stayed together. In SoA, all fields \( x \) are staying before all fields \( y \), and all fields \( y \) are straying before all fields \( z \). In SoAoS, the fields \( x \) and \( y \) are organized like in AoS, while all fields \( x \) and \( y \) are staying before all fields \( z \).

### 2.5 Automatic Data Layout Selection

For a GPU kernel, the best data layout can be hard to find, as the number of possible data layouts grows exponentially with the number of data fields. For example, suppose there are \( N \) fields, then the number of different SoAoS layouts is the \( N \)th Bell number \( B(N) \), where \( B(20) \) is already over 50 trillion [14]. Furthermore, finding the best data layout has been proven to be NP-hard [5, 6], which means brute forcing the best data layout is not feasible for kernels accessing a large number of fields.

In recent years, automatic data layout selection has been studied on GPUs [3, 4, 15]. This body of work identifies the benefits of using SoA for memory coalescing and using AoS for cache reuse, and proposes polynomial time algorithms to approximate the best data layout. Most of these approaches use a distance-based approach which starts from SoA, and merges fields that are accessed within a predefined instruction distance until hitting a stopping criteria. More details of existing distance-based approaches are discussed in Section 6.4.

However, none of these approaches provides a mathematical cost function to minimize, which makes objective analysis of these approaches difficult. This motivates us to develop a comprehensive cost analysis mechanism for data layout selection on GPUs. In this project, we first propose a cost function based on data layout and kernel memory accesses. Then, we provide an algorithm that enables cost estimation for
a given data layout at compile time. Our goal is to develop an accurate cost function and static cost estimation algorithm, so that data layouts with a lower estimated cost will have a smaller dynamic cost and lower running time.
Chapter 3

Dynamic Cost Functions

3.1 Assumptions

Since a data layout specifies the memory organization of a program’s data, the selection of the data layout will only affect the efficiency of the memory access operations.

Our cost function is not intended to approximate the execution time of a kernel, but evaluate the efficiency of the data layout. Therefore, we make the following assumptions before introducing our cost function.

1. We will only consider memory access operations in a kernel, and ignore the GPU’s ability to overlap computation with communication. The main reason is that the choice of the data layout will only affect the efficiency of memory access operations. This assumption will highly simplify the design of a cost function, and make static cost estimation feasible. The only drawback that we see from this approach is a benign one: our cost function may differentiate between two data layouts even when memory costs are negligibly small in both cases. We will consider the impact of other instructions in the future work.

2. Moreover, we only consider the L1 and L2 cache for the memory locality. On current Nvidia GPUs, the shared memory and the L1 cache share the same piece of on-chip memory, hence have the same latency. The only difference is that the shared memory is utilized explicitly, and the L1 cache is utilized implicitly. Thus, under our cost model, we can treat the shared memory like the L1 cache.
More detailed studies about shared memory and other on-chip special-purpose memories are left for future work.

3.2 Design of the Cost Functions

To emulate the real cost of a data layout, we build our cost function based on knowledge of the GPU memory hierarchy. In particular, since the GPU caches play an important role in data layout selection, our cost function should reflect the cache utilization of the data layout. At a high level, the cost function multiplies the number of global transactions that are likely to target L1, L2, and DRAM with the overhead of accessing those memory hierarchy levels and then sums up the product terms.

To build up our cost function, we first define the cost coefficient as the overhead of accessing different memory hierarchies. We define $W_{L1}$, $W_{L2}$ and $W_{DRAM}$ as the costs for accessing the L1 cache, accessing the L2 cache, and accessing the DRAM, respectively. Since off-chip memory accesses take significantly more time than on-chip memory accesses, $W_{L1}$ is usually much smaller than $W_{L2}$ and $W_{DRAM}$. Furthermore, hitting in the L2 cache is still preferred relative to accessing the DRAM.

Then we use $N_{read}$ and $N_{write}$ to denote the total number of global memory transaction requests made during the execution of the kernel. Moreover, we define $R_{L1,read}$, $R_{L2,read}$ and $R_{L2,write}$ as the overall L1 hit ratio for read requests, the overall L2 hit ratio for read requests, and the overall L2 hit ratio for write requests.

Then we define our cost function according to the formula below. Our approach is similar to the performance metrics defined in PerfExpert [16].

$$\text{Cost}_{\text{read}} = W_{L1} \times N_{read} \times R_{L1,read} + W_{L2} \times N_{read} \times (1 - R_{L1,read}) \times R_{L2,read} + W_{DRAM} \times N_{read} \times (1 - R_{L1,read}) \times (1 - R_{L2,read})$$

$$\text{Cost}_{\text{write}} = W_{L2} \times N_{write} \times R_{L2,write} + W_{DRAM} \times N_{write} \times (1 - R_{L2,write})$$
Total Cost = Cost_{read} + Cost_{write}

All terms in the above formulas are quantitative values, and the units will be cycles if the cost coefficients are approximated by the latency. They can be retrieved from GPU profiling metrics when executing the kernel. Two costs can be compared, and under our assumption, for a GPU kernel, the data layout with smaller cost will have a shorter execution time. In the native implementation, we use constants to represent the costs. However, we also introduce a vector representation to represent the cost in the GPU kernels with dynamic length loops, in order to make the estimation more accurate. More details about vector based representation will be discussed in Section 4.8.

3.3 Deriving the Dynamic Cost from Profiling Metrics

We can calculate the dynamic cost from the GPU profiling metrics when executing the kernel. This offers a quantitative and precise target to validating our cost estimation algorithm described in Chapter 4, while the goal of data layout selection is to find out the data layout which minimizes the total cost. We use constants to represent the dynamic cost because the loop trip count will be reflected from the profiling metrics.

The cost coefficients $W_{L1}$, $W_{L2}$ and $W_{DRAM}$, have to be estimated in advance for different GPU architectures. To simplify the cost estimation, we use the latencies of memory hierarchies to estimate the cost coefficient, and more details about estimating the cost coefficients will be discussed in Section 4.3.

One challenge when calculating the dynamic cost is we cannot retrieve the overall L2 hit ratio for write requests, $R_{L2,write}$, from the profiling metrics. Fortunately, under our assumption, most of the global memory write operations are initiated by the memory write operations in the GPU kernel. Thus, we can use the profiling
metric dram_write_transactions to approximate the number of memory writes that reach the global memory, which can be used for deriving the value of $R_{L2,\text{write}}$.

All other terms, $N_{\text{read}}$, $N_{\text{write}}$, $R_{L1,\text{read}}$, and $R_{L2,\text{read}}$, can be directly obtained from the profiling metrics. Table 3.1 shows a summary of how to get the value of each term in our cost function from the corresponding profiling metrics.

<table>
<thead>
<tr>
<th>Term</th>
<th>Profiling Metric</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_{\text{read}}$</td>
<td>glt_transactions</td>
<td>Integer</td>
</tr>
<tr>
<td>$N_{\text{write}}$</td>
<td>gtt_transactions</td>
<td>Integer</td>
</tr>
<tr>
<td>$R_{L1,\text{read}}$</td>
<td>l1_cache_global_hit_rate</td>
<td>Float, between 0 and 1</td>
</tr>
<tr>
<td>$R_{L2,\text{read}}$</td>
<td>l2_l1_read_hit_rate</td>
<td>Float, between 0 and 1</td>
</tr>
<tr>
<td>$R_{L2,\text{write}}$</td>
<td>1 - dram_write_transactions / l2_l1_write_transactions</td>
<td>Float, between 0 and 1</td>
</tr>
</tbody>
</table>

Table 3.1: Deriving the value of the terms in the cost function from the profiling metrics.

Once we have the values of the terms in the cost function, we can compute the dynamic cost. In order to use the dynamic cost as a criterion of data layout selection, we should keep the input data unchanged when executing the kernel with different data layouts. Depending on the actual scheduling of the GPU program execution, there might also be small differences in execution time between different runs of the same kernel with the same input data. In general, this process can be very labor-intensive, which is why we propose static estimation in the next chapter.

### 3.4 Illustration

In this section, we illustrate the calculation of our cost function using the two-field vector addition example with AoS and SoA layouts. For vectors with length $N$, we will create $N$ threads in total, and inside the GPU kernel, each thread calculates the sum of one pair of data entries in the vectors. The following CUDA code shows sample
kernels with both data layouts.

```c
__global__ void vectorAdditionAoS (MyAoS *inputStruct, float *result) {
    const int tid = blockIdx.x * blockDim.x + threadIdx.x;
    float tempX = inputStruct[tid].x;
    float tempY = inputStruct[tid].y;
    result[tid] = tempX + tempY;
}

__global__ void vectorAdditionSoA (MySoA *inputStruct, float *result) {
    const int tid = blockIdx.x * blockDim.x + threadIdx.x;
    float tempX = inputStruct->x[tid];
    float tempY = inputStruct->y[tid];
    result[tid] = tempX + tempY;
}
```

<table>
<thead>
<tr>
<th>Term</th>
<th>AoS</th>
<th>SoA</th>
</tr>
</thead>
<tbody>
<tr>
<td>N_{read}</td>
<td>527351</td>
<td>267708</td>
</tr>
<tr>
<td>N_{write}</td>
<td>131628</td>
<td>133798</td>
</tr>
<tr>
<td>R_{L1,read}</td>
<td>50.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>R_{L2,read}</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>R_{L2,write}</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>Total Cost</td>
<td>395567175.5</td>
<td>401506000</td>
</tr>
<tr>
<td>Execution Time (s)</td>
<td>0.781</td>
<td>0.784</td>
</tr>
</tbody>
</table>

Table 3.2: The value of the profiling metrics, the dynamic cost, as well as the execution time of the two-field vector addition example on an Nvidia Fermi GPU.

We ran both kernels on an a Fermi GPU (Nvidia Tesla M2050). Both L1 and L2 caches were enabled on this GPU. The detailed configuration of the GPU can be found in Section 5.2. The vector length was set to 4194304 (i.e. 2^{22}), and the data in the vectors were randomly generated. Following the estimation in Section 4.3, we let the cost coefficients W_{L1} = 1, W_{L2} = 30 and W_{DRAM} = 100.

Table 3.2 shows the profiling metrics, the dynamic cost, as well as the execution time of both kernels. From the table, we can see the AoS layout has twice number
of $N_{\text{read}}$ than the SoA layout, as the read of fields $x$ and $y$ were coalesced only in the SoA layout. However, the AoS layout has better cache hit ratio since the data fields $x$ and $y$ were declared in adjacent memory location for the same array index. As a result, the AoS and SoA layouts have similar estimated costs, as well as similar execution times.
Chapter 4

Static Cost Estimation

Although we can derive the dynamic cost from executing the kernel, running the kernel multiple times for different layouts can be expensive, and the dynamic cost may be highly affected by the input data. This motivates us to develop a static cost estimation algorithm that can estimate the cost at compile time. The static cost estimation can also be a good foundation for developing future automatic data layout selection tools.

To estimate the cost from a given data layout at compile time, we have to estimate the cost of different memory access operations separately and sums up the individual costs. To compute the cost of a memory access operation, we have to estimate the following terms for that operation, and we use the $\sim$ symbol to refer to estimated values:

- Cost coefficient: $\tilde{W}_{L1}$, $\tilde{W}_{L2}$, and $\tilde{W}_{DRAM}$.
- Number of transaction requests: $\tilde{N}_{\text{read}}$, $\tilde{N}_{\text{write}}$.
- Cache hit ratio: $\tilde{R}_{L1,\text{read}}$, $\tilde{R}_{L2,\text{read}}$ and $\tilde{R}_{L2,\text{write}}$.

4.1 Assumptions

It is difficult to estimate the behavior of the GPU cache at compile time because both L1 and L2 caches are shared by many warps, and the scheduling of warps is not predictable at compile time. Moreover, our cost model is not intended to estimate the
actual running time, but rather to determine whether a given data layout is better than another in terms of the total cost of global memory accesses. Thus, we make the following assumptions.

1. Both L1 and L2 cache adopt LRU cache replacement policy, as most modern GPUs use LRU or pseudo-LRU for both L1 and L2 caches [17].

2. We assume all threads are running in lockstep as a large, cross-SM warp, i.e. they are executing the same instruction concurrently, before moving on to the next instruction. The reason for making this assumption is that it is hard to determine the order of warp execution at compile time. Although this assumption may overestimate the cache hit ratio, it is a reasonable way to estimate cache behavior at compile time.

4.2 Basic Terminology for Accessing Global Memory

Before describing our cost estimation algorithm, we first define some basic terminology related to accessing global memory. Although some of these terms may have been mentioned in previous studies on data layout selection, we provide our own definitions to ensure our estimation algorithm is unambiguous. The terms in this section are not limited to SoAoS selection, although we illustrate these terms using SoAoS.

4.2.1 General Structure of SoAoS

In general, any SoAoS layout has an equivalent declaration as a combination of different AoSs, while each AoS Arr can be declared in the following format.

```c
typedef struct {
    type1 x;
    type2 y;
```
struct StructType {
  type3 z;
  ...
} StructType;
StructType Arr[Len];

The type here can be any 1, 2, 4, 8 or 16-byte data type. There will be some padding bytes to ensure the starting address alignment of every N-byte field is a multiple of N bytes. StructType may have one or more fields inside.

Then for each access to the field x inside the array Arr, where Arr is an array of StructType, we define the following terms.

4.2.2 Block Size, Grid Size, and Thread ID

For a GPU kernel, the block size is the number of threads inside each CUDA thread block, and the grid size is the number of CUDA thread blocks in total. For each thread, the thread ID (tid) is its one-dimensional rank inside the global thread pool.

For example, when launching a CUDA kernel, the programmer has to specify the block size and the grid size as <<<block size, grid size>>>, and inside the kernel, the thread ID can be fetched using the statement blockDim.x * blockIdx.x + threadIdx.x.

4.2.3 Structure Size

The structure size of a structure is the number of bytes of the structure including the padding bytes. The structure size is the actual amount of memory space the structure occupies.

For example, consider the following structure MyType. There will be 3 padding bytes after w to ensure the starting address of y is multiple of 4. Similarly, there will be 1 padding byte after y to ensure the starting address of z is multiple of 2. Thus,
including the padding bytes, the structure size of MyType is \((1 + 3 + 4 + 1 + 1 + 2) = 12\) bytes.

```c
typedef struct {
    char w; // 1 byte
    int x; // 4 bytes
    char y; // 1 byte
    short z; // 2 bytes
} MyType;
```

### 4.2.4 Array Index

The array index for accessing data field x using AoS as Arr[i].x, or using SoA as Arr.x[i], will both be i. Array indices can be constants, variables, mathematical expressions with variables, or even functions of variables. For example, Arr[i*j].x and Arr[foo(i+j)].x are allowed in the cost estimation, although we have to conservatively handle these cases.

### 4.2.5 Stride

The stride is defined between a pair of adjacent threads in the same warp, where a pair of adjacent threads have their thread IDs differ by 1. For a global access to data field x, the stride between a pair of adjacent threads in the same warp is the difference between the two memory addresses of the field x these two threads are accessing. The value of the stride depends on the difference of the array indices accessed between adjacent threads, as well as the structure size of the structure field x belongs to. Strides can be different for different pairs of adjacent threads.

The following formula shows how to calculate the stride between thread tid and tid + 1.
indexDiff = abs(array_index(tid) - array_index(tid + 1))

stride(tid, tid + 1) = indexDiff * sizeof(StructType)

Sometimes, indexDiff cannot be computed at compile time, especially when array indices are non-analyzable functions of the thread ID. In such cases, we would say the stride cannot be computed at compile time, or the stride is undefined for static cost estimation.

Consider the two-field vector addition example in Section 3.4 again. We can see if using AoS, both accesses to x and y has array_index(tid) = tid, thus indexDiff will be 1 for all adjacent threads. Thus all strides will be indexDiff * sizeof(MyAoS) = 8 bytes. On the other hand, if using SoA, both accesses to x and y also has array_index(tid) = tid, then indexDiff will also be 1 for all adjacent threads. Thus all strides will be indexDiff * sizeof (int) = 4 bytes.

Figure 4.1 shows another example when strides are different. Suppose A is an AoS with structure size equals to 8 bytes, and all threads are accessing A[tid * tid].x. Then since Thread 1 is accessing A[1].x, and Thread 2 is accessing A[4].x, the stride between Thread 1 and Thread 2 will be (4 - 1) * 8 = 24 bytes. The strides between other pairs of threads can be computed in a similar way. We can see the all strides are different between different pair of threads.

Figure 4.1: One example of different strides.
4.2.6 Instruction Distance

We define the *L1 and L2 instruction distance* between two memory access instructions $I_1$ and $I_2$, as the number of unique memory locations accessed by all threads sharing the L1 or L2 cache between $I_1$ and $I_2$. According to our second assumption in Section 4.1, different threads will have the same contribution to the instruction distance. Moreover, fetching a field into the cache will also fetch the fields declared in the structure if they fall in the same cache line, and these fields will also contribute to the unique memory locations even if they were not accessed.

Then, the L1 instruction distance between two memory access instructions can be calculated as:

$$\text{#_blocks_per_SM} \times \text{block_size} \times \text{unique memory locations in between (inclusively) in one thread}$$

And the L2 instruction distance between two memory access instructions can be calculated as:

$$\text{grid size} \times \text{block size} \times \text{unique memory locations in between (inclusively) in one thread}$$

Based on our LRU assumption in Section 4.1, there will be no benefit of cache reuse between accesses with instruction distance larger than the cache size.

For example, for the following accesses to $\text{Arr1}[i].x$ and $\text{Arr1}[i].y$, where $\text{Arr1}$ is an array of structure containing integer fields $x$ and $y$, and $\text{Arr2}$ is an array of structure containing integer fields $w$ and $z$. All fields are 4-byte data type.

$$\ldots = \text{Arr1}[i].x \hspace{1em} \text{// Instruction A}$$
$$\ldots = \text{Arr2}[i].z$$
$$\ldots = \text{Arr2}[i].w$$
Suppose the grid size is 1024, the block size is 256, and the number of blocks per SM is 8. Then between Instruction A and B, the L1 instruction distance is $8 \times 256 \times 4 \times 4 = 32768$ bytes, and the L2 instruction distance is $1024 \times 256 \times 4 \times 4 = 4194304$ bytes.

4.2.7 Index Accordance

In a thread, for accesses to different fields $x$ and $y$ which are candidates for cache reuse, we are less interested in the actual difference between their array indices. Instead, we are more interested if these two instances of field $x$ and $y$ will be in the same cache line. Thus, we say two accesses are $L1_{beneficial}$ if the elements accessed can be proven to be in the same L1 cache line, and they are $L2_{beneficial}$ if the elements accessed can be proven to be in the same L2 cache line. In our approach, index accordance is only meaningful for fields declared in the same structure.

For example, the following accesses to $x$ and $y$ are neither $L1_{beneficial}$ nor $L2_{beneficial}$, as we cannot guarantee that $Arr1[i].x$ and $Arr1[N - i].y$ will be in the same L1 or L2 cache line.

4.3 Estimating the Cost Coefficient

Because the GPU has the ability to overlap memory accesses with computational instructions, precisely defining the cost of different types of memory accesses can be difficult. However, since arithmetic operation latency on the GPU usually consumes
significantly fewer cycles than memory access operations, a conservative approach is to assign cost coefficients based on memory access latency normalized to a hit in the L1 cache. Unfortunately, there is no official release of latencies of the GPU memory hierarchy. Table 4.1 shows some approximated latency values for Fermi from running microbenchmarks [18, 19]. The microbenchmarks mainly record the time for repeating memory accesses that hit the L1 cache, hit in the L2 cache and miss in the L2 cache. By comparing the relative accessing time, we can estimate the cost coefficients for the GPU architecture.

<table>
<thead>
<tr>
<th>Memory Hierarchy</th>
<th>Latency (Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache</td>
<td>10</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>300</td>
</tr>
<tr>
<td>DRAM</td>
<td>1000</td>
</tr>
</tbody>
</table>

Table 4.1 : Latencies for the memory hierarchy in an Nvidia Fermi GPU.

Thus, we set $\tilde{W}_{L1} = 1$, $\tilde{W}_{L2} = 30$, $\tilde{W}_{DRAM} = 100$ for Fermi as relative costs. For different GPU architectures, we set $\tilde{W}_{L1}$, $\tilde{W}_{L2}$ and $\tilde{W}_{DRAM}$ in a similar way.

### 4.4 Estimating the Number of Transaction Requests

For each memory access instruction, the stride is the only attribute needed to estimate the number of transaction requests. In particular, there are three common cases:

- If all strides within a warp equal 4 bytes, we say the access has a *unit stride*, or the access is *fully coalesced*. In this case, if the memory address of the field to be requested from the first thread is a multiple of 128 bytes, then inside a warp, there will be only one global memory transaction. Moreover, if all strides $= 0$ inside a warp, i.e. all threads are accessing the same element, then only
one global memory transaction will be needed for all threads inside the same warp as well.

- If all strides equal to $N \times 4$ inside a warp, for some $N$ where $1 < N < 32$, then inside a warp there will be $N$ 128-byte transactions needed.

- If all strides can be proven to be larger than or equal to 128 bytes, where they do not have to be equal among different warps, then we say the access is *fully non-coalesced*. In such case, inside a warp, 32 separate global memory transactions will be needed. Moreover, if the array index is independent of the thread id, and the stride cannot be analyzed at compile time, then we will regard the access as fully non-coalesced.

Unfortunately, in some cases, the stride cannot be known at compile time since the array indices may be functions of runtime variables. In these cases, we make the conservative estimation that all dynamic strides are larger than 128 bytes.

Finally, inside a warp, by examining all the strides as well as all the relative memory address, we would be able to determine the number of unique memory segments requested by the warp. Then next step of cost estimation is to determine which memory hierarchies these transaction requests finally go to, i.e. whether they will hit the L1 or L2 cache, or they will go to the DRAM.

### 4.5 Estimating the Cache Hit Ratio

In any computer architectures, a requested data will be hit in the cache if and only if both of the following conditions hold:

- Some data of the same cache line was accessed before, so that the requested data was loaded into the cache line.
• The cache line was not replaced.

Thus, we use the following rule to estimate the cache hit ratio: for an access to field $y$ after an access to field $x$, if and only if their instruction distance is not greater than the L1 or L2 cache size, and the two accesses are L1_beneficial or L2_beneficial, then $y$ may be a hit in the L1 or L2 cache.

To determine if the index accordance is satisfied, we have to look at the array indices, as well as the structure size. Suppose we have two accesses to $\text{Arr}[i].x$ and $\text{Arr}[j].y$. First, if the difference of array indices, i.e. the absolute value of $i$ minus $j$, cannot be known at compile time, then we will say the index accordance is not satisfied, although this may miss some opportunities for cache reuse. Otherwise, we look into such difference times the size of the struct of the array $\text{Arr}$.

\[(|i - j| + 2) \times \text{sizeof(dataType(Arr))}\]

Then, we say the L1 or L2 index accordance will be satisfied, if and only if the above value is less than or equal to the L1 or L2 cache line size.

An access to the field $x$ will be hit in the L1 or L2 cache if there exists some field $y$ declared in the same structure with $x$, where there is an access to the field $y$ before the access to the field $x$ within the L1 or L2 instruction distance, and the L1 or L2 index accordance was satisfied. Otherwise, DRAM will be accessed for accessing field $x$. Among different threads inside the same warp, whether the access will be hit in the L1 or L2 cache may be different. Finally, the cache hit ratio can be retrieved by considering the cache hit or miss of all threads.
4.6 Estimating the Number of Blocks per SM

When estimating the cache hit ratio, we have to compute the L1 instruction distance between a pair of memory access operations. According to the formula of the L1 instruction distance, the number of blocks per SM can not be directly known at compile time. Thus, in order to calculate the L1 instruction distance, we have to firstly estimate the number of blocks per SM. When executing a kernel, the number of blocks per SM is restricted by the following factors:

- The maximum number of thread blocks an SM can have for the GPU architecture, $max_{\#\text{blocks}\_\text{per}\_\text{SM}}$. 

- The maximum number of threads an SM can have for the GPU architecture, $max_{\#\text{threads}\_\text{per}\_\text{SM}}$. For any kernel, we always have

\[
\text{block}_\text{size} \times \#\text{blocks}\_\text{per}\_\text{SM} \leq \text{max}_{\#\text{threads}\_\text{per}\_\text{SM}}
\]

- The number of registers an SM has for the GPU architecture, $\#\text{registers}\_\text{per}\_\text{SM}$, which can be retrieved using `cudaGetDeviceProperties` API [20]. Moreover, when compiling a kernel, we can retrieve the number of registers requested by the kernel, $\#\text{registers}\_\text{per}\_\text{thread}$. For example, we may pass the `-Xptxas=-v` option if using the `nvcc` compilation command [21]. Finally, we should have

\[
\text{\#registers}\_\text{per}\_\text{thread} \times \text{block}_\text{size} \times \#\text{blocks}\_\text{per}\_\text{SM} \leq \text{\#registers}\_\text{per}\_\text{SM}
\]

As a summary, we can use the following formula to estimate the number of blocks per SM for a kernel.
#_blocks_per_SM = max(1, min(#_blocks_per_SM, max(#_threds_per_SM / block_size, #_registers_per_SM / #_registers_per_thread / block_size))

4.7 Estimating the Total Cost

To estimate the total cost, we must have access to the kernel, a given data layout, as well as:

1. Grid size and block size. We also provide default values for them if they are not explicitly specified.

2. The number of threads per warp. For all current Nvidia GPUs this value is 32.

3. L1 cache size. For Fermi the L1 cache size can be configured as 16KB or 48KB, and for Kepler the L1 cache size can be configured as 16KB, 32KB or 48KB. We set this value to be 0 if the L1 cache is not enabled.

4. L1 cache line size. For all current GPUs this value is 128 Bytes.

5. L2 cache size. The L2 cache size for Fermi is 768KB, and for Kepler it is 1536KB.

6. L2 cache line size. For all current GPUs this value is 32 Bytes.

Figure 4.2 shows the dependency graph of the attributes for cost estimation, which guides the order of the cost estimation. During cost estimation, the attributes should be estimated according to the topological order of this graph.

Theoretically, to compute the total amount of memory transactions, we would need to consider each execution of every memory access instruction. However, in
practice, we usually cannot retrieve how many times each memory access instruction will be executed at compile time, especially when the instruction is in a loop with dynamic length. Thus we may need to give an estimation of the value of dynamic loop lengths. In practice, our give the estimation of the length of all dynamic loops as 100. Then we can unroll all the loops, and transform the kernel into an extend basic block.

In the transformed code, for each global memory access instruction, we estimate the cost in the granularity of warps. We use the approaches in Section 4.4 to estimate the number of global transaction requests, and use the rules in Section 4.5 to estimate the cache hit ratio. Then we multiply the number of transactions hit in the L1 cache, hit in the L2 cache, and go to DRAM with their corresponding cost coefficient, which will be estimated using the rules introduced in Section 4.3. Finally, the cost of the memory access instruction can be calculated by summing up the costs from all warps.

Finally, adding up the cost of all global memory accesses, we will get the total estimated cost.

The pseudo-code to estimate the cost is given as follows:

```c
void add (Queue Q, vector index, int size)
```
if (Q.size == 0) return; // In case L1 cache is not enabled
if (Q.size == Q.sizeLimit) Q.dequeue(); // Cache replacement
Q.enqueue(index - index % size, size);

bool hit (Queue Q, vector index)
{
    return whether Q contains some (y, size) where y < index < y + size;
}

vector costEstimation (int N)
// N is the number of times to unroll the loop with variable length
{
    totalCost = new vector(0, 0, ..., 0);
    Queue L1[gridSize] = {empty queue};
    // One L1 queue for each block
    Queue L2 = empty queue;
    // One L2 queue in total

    Unroll each loop into one basic block;

    for each global memory access instruction I
    {
        vec = frequency vector of I;
        S = sizeof (StructType);
        for each block
            for each warp
            {
                index = new int[32];
                for each thread inside current warp
                {
                    tid = blockID * blockSize + threadID;
                    if (the array index can be determined at compile time)
                        index[tid % 32] = (value of array index given tid) * S;
                    else
                        index[tid % 32] = -1;
                }
                sort (index);
                i = 0;
                while (index[i++] == -1 && i<32)
                    // skip the indices which can't be determined
                {
                    totalCost += Wdram * vec;
                    add(L1[blockID],[-1,-1]);
                }
            }
        }
    }
}
35

add(L2,-1,-1);
i++;
}
while (i < 32)
{
    if (hit(L1[blockID], index[i]))
        totalCost += Wl1 * vec;
    else {
        if (hit(L2, index[i])) {
            totalCost += Wl2 * vec;
            add(L1[blockID],index[i],32);
        } else {
            totalCost += Wdram * vec;
            add(L2,index[i],32);
            N = 32, 64 or 128 bytes
            // depends on whether the access operation is reduced
            add(L1[blockID],index[i],N);
        }
    }
    i++, until i>=32 or index[i] is in a different memory segment;
}
return totalCost;

In practice, given two different data layouts, our cost estimation algorithm computes the cost of each data layout. The layout with lower cost will potentially have a better memory access efficiency, hence smaller execution time.

4.8 Handling Dynamic Length Loops

Using a constant to approximate the length of dynamic loops is a standard approach in static cost estimation. However, in our experience, this approach can be less effective when some loops have dynamic lengths and others have lengths that are compile-time constants. Thus, as in big-O analysis, where variable terms are assumed to be more significant than constant terms, we take an approach in which loops with variable
lengths are considered to be more significant than loops with constant lengths.

We start by defining the complexity degree (CD) of memory access instruction $I$, as the number of loops enclosing $I$ with variable loop length. We also define the complexity vector (CV) of memory access instruction $I$, enclosed in $n$ loops, as a vector with a 1 at position $\text{CD}(I)$, and zeroes in all other positions. For a kernel, the length of the complexity vector equals to the depth of most nested loop plus one, while each term represents different possible complexities.

For example, consider the following loop nest in which $M$ and $N$ are unknown at compile time:

```cpp
for (i = 0; i < M; i++)
    for (j = 0; j < 128; j++)
        for (k = 0; k < N; k++)
            {
                ...
                a[index].x ++; // Instruction A
                ...
            }
```

Then $\text{CD}(A)$ will be 2, since there are two loops with unknown length enclosing $A$. Suppose the most nested loops in the kernel has depth 3, then $\text{CV}(A)$ will be $(0, 0, 1, 0)$ with a 1 in position 2, thereby representing a quadratic complexity.

During cost estimation, for each memory access instruction, once we have computed its estimated cost, we multiply it by the complexity vector for that instruction. The resulting vector is called the cost vector. Two cost vectors can be added using standard vector addition, and we use this way to sum up the total cost.

Finally, we have to be able to determine if one cost vector is larger than the other. Since the rightmost entry of a cost vector is the most significant entry, we say that $(X_0, X_1, ..., X_n) > (Y_0, Y_1, ..., Y_n)$, if and only if, there exists some $0 < m <= n$, $(X_m > Y_m)$ and $(X_{m+1} = Y_{m+1})$ and $(X_{m+2} = Y_{m+2})$ ... and $(X_n = Y_n)$. 
Our vector-based representation will prefer data layouts which focus on optimizing accesses within dynamic length loops, which will have a higher chance of being executed more frequently. For example, for the following code, Instruction A and B are the only global memory access instructions, and the value of M and N cannot be known at compile time. Suppose the grid size is 100, and the block size is 256. Assume that we have two data layouts: Layout_1 and Layout_2. To make the example simple, suppose using Layout_1, for each warp, the cost of every execution of Instruction A is always 1 and the cost of every execution of Instruction B is always 3. Similarly, if using Layout_2, for each warp, the cost of every execution of Instructions A is always 3 and the cost of every execution of Instruction B is always 1.

```c
for (i = 0; i < M; i++)
{
    for (j = 0; j < 128; j++)
    {
        ... = a[index1].x; // Instruction A
    }
    for (k = 0; k < N; k++)
    {
        ... = a[index2].y; // Instruction B
    }
}
```

First, we see that the complexity degrees for instructions A and B are $\text{CD}(A) = 1$ and $\text{CD}(B) = 2$, yielding $\text{CV}(A) = (0, 1, 0)$ and $\text{CV}(B) = (0, 0, 1)$.

Then, if we assume the loop i and k will execute 100 times, the cost vector for Layout_1 will be $100 \times 128 \times 100 \times 256 \times 1 \times (0, 1, 0) + 100 \times 100 \times 100 \times 256 \times 3 \times (0, 0, 1) = (0, 327680000, 768000000)$.

And the cost vector for Layout_2 will be $100 \times 128 \times 100 \times 256 \times 3 \times (0, 1, 0) + 100 \times 100 \times 100 \times 256 \times 1 \times (0, 0, 1) = (0, 983040000, 256000000)$.

Since $(0, 327680000, 768000000) > (0, 983040000, 256000000)$, Layout_2, which
better accommodate Instruction B, will be preferred by our cost estimation algorithm. The reason is we believe Instruction B will be executed more frequently than Instruction A in practice. From the example, we can see that even we approximate the dynamic loop length with small constants (100), which is smaller than constant loop length (128), dynamic length loops can still be accommodated prior to constant length loops.
Chapter 5

Performance Evaluation

5.1 Performance Evaluation Methodology

The performance evaluation is done in the following way. First, for each benchmark, we propose some reasonable data layouts, including AoS and SoA. Then we compute the cost for each data layout using our cost estimation algorithm. After that, we run the benchmark written in each of the chosen data layouts, and record the total running time and all profiling metrics related to global memory access operations. We can then compute the dynamic cost using the profiling metrics and the relative costs \( \tilde{W}_{L1} \), \( \tilde{W}_{L2} \) and \( \tilde{W}_{DRAM} \). Finally, we compare the actual running time with the estimated cost and the dynamic cost. We expect the layout with lower estimated cost will have smaller dynamic cost, as well as smaller execution time.

5.2 Machine Configuration

We run our benchmarks on both Fermi and Kepler GPUs. The Fermi GPU we used was an Nvidia Tesla M2050 GPU [22]. The Kepler GPU we used was an Nvidia Tesla K20c GPU [23]. In this generation of Kepler GPU, the L1 cache is not enabled for caching global memory transactions. Table 5.1 shows the attributes of the machines used during performance evaluation.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Tesla M2050</th>
<th>Tesla K20c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CUDA Cores</td>
<td>448</td>
<td>2496</td>
</tr>
<tr>
<td>Warp Size</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>L1 Cache Size</td>
<td>64KB</td>
<td>0KB</td>
</tr>
<tr>
<td>L1 Cache Line Size</td>
<td>128 Bytes</td>
<td>Not applicable</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>768KB</td>
<td>1536KB</td>
</tr>
<tr>
<td>L2 Cache Line Size</td>
<td>32 Bytes</td>
<td>32 Bytes</td>
</tr>
</tbody>
</table>

Table 5.1: Attributes of the GPUs used for evaluation.

5.3 Benchmarks

To evaluate the effectiveness of our proposed strategy for computing cost models for data layouts, we applied it to the following benchmarks: K-means, N-body and LavaMD. All these benchmarks have their unique feature to be studied in data layout selection problems.

5.3.1 K-means

K-means is a clustering algorithm in data mining. It keeps assigning the points to the cluster, recomputes the centroid of the new clusters, until reaching a fixed point. Inside the kernel, although the fields feature and clusters field are accessed within a short instruction distance, however, the index accordance was not satisfied. Thus using AoS will not be beneficial from the cache reuse. For this benchmark, we choose AoS, SoA, and SoAoS which merges the fields feature and clusters.

5.3.2 N-body

N-body is the physics algorithm that predicts the motion of each object by considering the forces from all other objects. Except loading and storing the coordinates of the current object, all of the other global accesses are accessing the same element at each time. Thus most of the accesses can be done within one transaction in every warp.
We choose data layouts AoS, SoA, and two versions of SoAoS. The first SoAoS groups input fields \( x, y, z \) and output fields \( v_x, v_y \) and \( v_z \), and the second SoAoS groups \( x \) with \( v_x \), \( y \) with \( v_y \), and \( z \) with \( v_z \).

### 5.3.3 LavaMD

LavaMD simulates the relocation of particles in 3D space. The computation is performed on the granularity of cubes, where each cube has 26 surrounding neighbors. Since the grid computation will take the advantage of cache reusing, using AoS for each point might be more beneficial. We choose data layouts AoS, SoA and two versions of SoAoS. Like N-body, the first SoAoS groups all the dimensions of the input point, as well as the output point. The second SoAoS groups each dimension of the input and the output point.

### 5.3.4 Summary

Table 5.2 shows the benchmarks and data layouts we selected for our performance evaluation.

### 5.4 Performance Results

The reference code for our N-body benchmark was modified from Nyland’s Fast N-body implementation [24]. The reference code for our K-means and LavaMD benchmarks was modified from Rodinia’s implementation [25]. All the codes are written in CUDA. To simplify comparison of our results, all the loop lengths are given at compile time so that the estimated cost can be represented by a single value.

Figure 5.1 shows the performance results on Fermi, and Figure 5.2 shows the performance results on Kepler. For each benchmark and data layout, we present the
Table 5.2: Benchmarks and data layouts for our performance evaluation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Data Layout</th>
<th>Field Grouping</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-means AoS</td>
<td>{feature, clusters, membership}</td>
<td></td>
</tr>
<tr>
<td>K-means SoA</td>
<td>feature, clusters, membership</td>
<td></td>
</tr>
<tr>
<td>K-means SoAoS</td>
<td>{feature, clusters}, membership</td>
<td></td>
</tr>
<tr>
<td>N-body AoS</td>
<td>{x, y, z, vx, vy, vz}</td>
<td></td>
</tr>
<tr>
<td>N-body SoA</td>
<td>x, y, z, vx, vy, vz</td>
<td></td>
</tr>
<tr>
<td>N-body SoAoS 1</td>
<td>{x, y, z}, {vx, vy, vz}</td>
<td></td>
</tr>
<tr>
<td>N-body SoAoS 2</td>
<td>{x, vx}, {y, vy}, {z, vz}</td>
<td></td>
</tr>
<tr>
<td>LavaMD AoS</td>
<td>{d_rv_gpu.v, d_rv_gpu.x, d_rv_gpu.y, d_rv_gpu.z, d_fv_gpu.v, d_fv_gpu.x, d_fv_gpu.y, d_fv_gpu.z}</td>
<td></td>
</tr>
<tr>
<td>LavaMD SoA</td>
<td>d_rv_gpu.v, d_rv_gpu.x, d_rv_gpu.y, d_rv_gpu.z, d_fv_gpu.v, d_fv_gpu.x, d_fv_gpu.y, d_fv_gpu.z</td>
<td></td>
</tr>
<tr>
<td>LavaMD SoAoS 1</td>
<td>{d_rv_gpu.v, d_rv_gpu.x, d_rv_gpu.y, d_rv_gpu.z}, {d_fv_gpu.v, d_fv_gpu.x, d_fv_gpu.y, d_fv_gpu.z}</td>
<td></td>
</tr>
<tr>
<td>LavaMD SoAoS 2</td>
<td>{d_rv_gpu.v, d_fv_gpu.v}, {d_rv_gpu.x, d_fv_gpu.x}, {d_rv_gpu.y, d_fv_gpu.y}, {d_rv_gpu.z, d_fv_gpu.z}</td>
<td></td>
</tr>
</tbody>
</table>

result of the estimated cost, the dynamic cost and the execution time against the AoS layout. For the estimated cost and the dynamic cost, since the actual number can be very large, we give the relative ratio against the cost of the AoS layout. For the relative execution time, a value smaller than 1 denotes a slowdown against the AoS layout, and a value larger than 1 denotes a speedup against the AoS layout.

Since the cost of memory access operations is only part of the execution overhead, we do not expect the ratio of the cost to be approximately equal to the ratio of the execution time, but they are supposed to have a positive correlation. In other words, if the relative cost is smaller than 1, we expect the relative execution time also be smaller than 1, i.e. a speedup. Similarly, if the relative cost is larger than 1, we expect the relative execution time also be larger than 1, i.e. a slowdown.

From both figures, we can see our cost function and estimated cost vector can accurately predict the benefit among different data layouts on different GPU archi-
Figure 5.1 : Performance result on an Nvidia Fermi GPU, normalized to the AoS layout.

Figure 5.2 : Performance result on an Nvidia Kepler GPU, normalized to the AoS layout.
5.5 Discussion

Table 5.3 shows more detailed performance results in numbers. After analyzing the numerical results, we found that most of the errors between estimated cost and dynamic cost come from our assumption about warp scheduling. Moreover, ignoring the GPU’s ability to overlap computation with communication will also simplify the actual warp scheduling.

For example, for the LavaMD benchmark, the difference between the estimated cost and the dynamic cost is much larger than the other two benchmarks. We believe the reason is because there are more arithmetic operations in LavaMD, so the warp...
scheduling might be more complicated than our assumption. Furthermore, from the results, our cost estimation works better on Kepler than Fermi. We believe the reason is that Kepler has a faster processing unit than Fermi, so the ratio of cycles of arithmetic operations over memory access operations on Kepler will be smaller than Fermi. Thus, our assumption about ignoring the arithmetic operations will have less impact on Kepler.
Chapter 6

Related Work

6.1 Cost and Data Layout Analysis for CPUs

The cost analysis and data layout selection on CPUs have been studied for many years, and the existing works provide good references for our project. The design of our cost function and our cost estimation algorithm is based on the theory of cache reuse distance, as well as previous studies on modeling CPU memory accesses.

In particular, Keramidas et al. [26] and Beyls et al. [27] provided a detailed and theoretical analysis of cache reuse distance. Truong et al. [28] discussed the relationship between data layout and cache reuse, and designed the algorithm to optimize the data allocation. Zhong et al. [29] introduced the Reference Affinity Model, based on the cache reuse distance theory, to evaluate the impact of data layout on CPUs. Sharma et al. [30] developed an algorithm to regroup the data fields of CPU kernels based on the cache reuse distance. Although the memory hierarchies of CPUs are different from GPUs, these work still provide useful references for our project.

6.2 Cost Analysis for GPUs

General cost analysis for GPUs has been well studied for years. In particular, Govindaraju et al. [31] discussed modeling the cost of GPU cache in detail. Hong et al. [32] proposed an analytical model to estimate the execution time of GPU kernels and discussed the concept of memory coalescing in details. Zhang et al. [33] proposed a cost
model that considers the efficiency of global memory access together with the shared memory access and instruction pipeline. Nugteren et al. [34] and Tang et al. [35] used the reuse distance theory to model the performance of GPU caches. Wong et al. [36] studied the GPU memory hierarchies through microbenchmarking.

However, some of the above algorithms are making assumptions on the input program, and might tolerate small errors in the estimation. In contrast, we developed our cost function only based on the GPU memory hierarchy, and made significantly less assumption on the input program. Moreover, our cost estimation enables compile time estimation, which is especially good for the data layout selection problem.

6.3 Data Layout Analysis for GPUs

Currently, most of the data layout analysis algorithms for GPUs focus on the hybridization of arrays and structures. In particular, Mei et al. [37] discussed the benefit of different data layouts such as AoS, SoA, SoAoS, and hybrid data layouts such as array tiling inside SoAoS. They also identified the different data layout behaviors between single and double precision. Moreover, Cardenas-Montes et al. [38] provided the effect of data layouts on non-separable functions on GPUs. They provided detailed mathematical analysis on different GPU memory architectures such as caches, shared memory and registers. Furthermore, Hoshino et al. [39] compared the impact of data layouts on different architectures such as CPUs, GPUs and Intel Xeon Phi.

All these projects provided good references for modeling the data layout selection problems on GPUs. However, these projects either provide no mathematical cost model, or give some cost models based on the actual behavior of the GPU memory hierarchies with no accompanying static cost analysis models. This motivates us to build a predictive cost model which can accurately represent the cost of data layouts.
6.4 Automatic Data Layout Selection on GPUs

6.4.1 Distance Based Approaches

Most of the existing automatic GPU data layout selection projects use some sort of distance-based approach. Here the term distance refers to the instruction distance defined in Section 4.2. The main idea in these approaches is defining the instruction distance between two instructions as the amount of unique memory spaces between these two instructions, and the cost between two fields as the maximum instruction distance among all closest pair of instructions accessing those fields. Then the field pairs whose cost below a threshold are candidates to be merged into the same struct.

Kofler et al. [3] used a minimum spanning tree construction method to do the field partitioning based on the instruction distance, while they also enabled array tiling based on the profiling runs of the kernel with different tile sizes. The data layout selection algorithm built a control flow graph where each instruction corresponds to one node in the graph. The cost between nodes was defined to be the minimum amount of memory space between the corresponding instructions when executing the kernel. Then the distance between fields $f_1$ and $f_2$ are defined as the maximum cost between instructions $\text{MI}(f_1)$ and $\text{MI}(f_2)$, where $\text{MI}(f_1)$ and $\text{MI}(f_2)$ has data accesses to $f_1$ and $f_2$, respectively. Figure 6.1 shows the cost function of this work.

$$\delta(f_1, f_2) = \max\left(\max_{i \in \text{MI}(f_1)}\max_{j \in \text{MI}(f_2)} MD(i, j)\right)$$

Figure 6.1 : The cost function proposed by Kofler et al. Figure source: [3].
Majeti et al. [4] proposed a compiler framework called *Heterogeneous Habanero-C (H2C)*, which performs greedy approach to select the best SoAoS based on the instruction distance, while their work also inspected the data layout transformation between multiple kernels, and allows for data layout transformations between kernels. In their data layout selection algorithm, an affinity graph was created based on the distance between data fields and the frequency of each basic block. The cost function is shown in Figure 6.2.

\[
    w(e(a[i], b[i])) = \begin{cases} 
    0, & \text{if } \text{mem}(a[i], b[i]) > \text{cache-size} \\
    \text{freq}(B) \times \frac{1}{\log_2 \text{mem}(a[i], b[i])}, & \text{otherwise}
    \end{cases}
\]

Figure 6.2: The cost function proposed by H2C. Figure source: [4].

Compared to our cost analysis, existing distance based approaches make the same assumptions our cost model did:

1. LRU replacement policy for L1 and L2.
2. Threads are executing as in a big warp, as there is no difference among the formula to calculate the distance across different threads.
3. Ignoring the ability of GPU to overlap the computation with communication, as well as the possible utilization of shared memory.

However, existing distance based approaches make additional assumptions relative to our approach.

1. Most related works use one distance for the L1 and the L2 cache, or just ignore the benefit of the L2 cache. We define L1 and L2 instruction distances separately, as the threads sharing the L1 and the L2 cache are organized differently. Moreover, the benefits of hitting in the L1 and the L2 cache are also different.
2. Most related works do not look into the array indices of each access, or assume
the array indices always fall into the same cache line, i.e. they ignore the index
accordance defined in Section 4.2. This assumption is not true in general, and
it is sometimes unreasonable.

Without these assumptions, our cost analysis should be able to handle more gen-
eral cases more accurately than in past work.

6.4.2 Other Approaches

Beside distance based approaches, there are also several novel data layout selection
algorithms introduced by other studies. In particular, Weber et al. [40] proposed
an adaptive data layout selector based on the prediction function which predicts the
cache behavior of GPU. The prediction function takes small samples in the whole data
layout space and is able to approximate the data layout with best cache utilization.
Moreover, Dymaxion which developed by Che et al. [41], provided a series of APIs
which enables automatic data layout transformation on GPU.

However, although these approaches have achieved good performance in practice,
they did not provide a direct quantitative cost model that can be used for selecting
data layouts. Moreover, these approaches may have more limitations on the input
than distance-based approaches. For example, currently Dymaxion only works on
single-dimension linear memory and loop-based algorithms.
Chapter 7

Conclusions and Future Work

In this work presented in this thesis, we propose a novel cost analysis scheme for the data layout selection on GPUs. We build our cost function based on the cache hierarchy and memory access semantics of GPUs, to maximize the accuracy of the cost model. We also develop an algorithm to estimate the cost of a given data layout at compile time, which mainly simulates the LRU behavior of the GPU cache. We also introduce our novel cost vector representation to handle variable loop lengths at compile time. We test our cost estimation using selected benchmarks on Nvidia Fermi and Kepler GPUs. Our performance result shows our cost model can accurately predict the efficiency of a given data layout.

Currently, we are developing an automatic data layout selector for CUDA kernels. The data layout selector extends the existing distance based field merging algorithm, and it follows the cost model proposed in this thesis. A similar approach could be applied to develop a cost model for other architectures such as Intel Xeon Phi co-processor.
Bibliography


[38] M. Cárdenas-Montes and M. A. Vega-Rodríguez, “Effect of Data Layout in the

